project_notebook_sequence_analysis

April 26, 2019

1 Sequence Analysis with Python

Contact: Veli Mäkinen veli.makinen@helsinki.fi The following assignments introduce applications of hashing with `dict()` primitive of Python. While doing so, a rudimentary introduction to biological sequences is given. This framework is then enhanced with probabilities, leading to routines to generate random sequences under some constraints, including a general concept of *Markov-chains*. All these components illustrate the usage of `dict()`, but at the same time introduce some other computational routines to efficiently deal with probabilities.
The function `collections.defaultdict` can be useful.

1.1 DNA and RNA

A DNA molecule consist, in principle, of a chain of smaller molecules. These smaller molecules have some common basic components (bases) that repeat. For our purposes it is sufficient to know that these bases are nucleotides adenine, cytosine, guanine, and thymine with abbreviations `A`, `C`, `G`, and `T`. Given a *DNA sequence* e.g. `ACGATGAGGCTCAT`, one can reverse engineer (with negligible loss of information) the corresponding DNA molecule.

Parts of a DNA molecule can *transcribe* into an RNA molecule. In this process, thymine gets replaced by uracil (`U`).

1. Write a function `dna_to_rna` to convert a DNA sequence into an RNA sequence. For the sake of exercise, use `dict()` to store the symbol to symbol encoding rules. Create a program to test your function.

```
In [ ]: from IPython.core import page
        page.page=print # turn off the pager, so that %pycat prints inline

In [ ]: # exercise 1 DO NOT MODIFY THIS LINE
        %pycat part07-e01_dna_to_rna/src/dna_to_rna.py
```

1.1.1 Idea of solution

fill in

1.1.2 Analysation of results

fill in

1.2 Proteins

Like DNA and RNA, protein molecule can be interpreted as a chain of smaller molecules, where the bases are now amino acids. RNA molecule may *translate* into a protein molecule, but instead of base by base, three bases of RNA correspond to one base of protein. That is, RNA sequence is read triplet (called codon) at a time.

2. Consider the codon to amino acid conversion table in http://www.kazusa.or.jp/codon/cgi-bin/showcodon.cgi?species=9606&aa=1&style=N. Write a function `get_dict` to read the table into a `dict()`, such that for each RNA sequence of length 3, say `AGU`, the hash table stores the conversion rule to the corresponding amino acid. You may store the html page to your local src directory, and parse that file.

```
In [ ]: # exercise 2 DO NOT MODIFY THIS LINE
        %pycat part07-e02_rna_to_prot/src/rna_to_prot.py
```

1.2.1 Idea of solution

fill in

1.2.2 Analysation of results

fill in

3. Use the same conversion table as above, but now write function `get_dict` to read the table into a `dict()`, such that for each amino acid the hash table stores the list of codons encoding it.

```
In [ ]: # exercise 3 DO NOT MODIFY THIS LINE
        %pycat part07-e03_prot_to_rna/src/prot_to_rna.py
```

1.2.3 Idea of solution

fill in

1.2.4 Analysation of results

fill in

With the conversion tables at hand, the following should be trivial to solve.

4. Fill in function `dna_to_prot` in the stub solution to convert a DNA sequence into a protein sequence. You may copy-paste the result dictionaries from exercises 2 and 3. Run your program with `ATGATATCATCGACGATGTAG`.

```
In [ ]: # exercise 4 DO NOT MODIFY THIS LINE
        %pycat part07-e04_dna_to_prot/src/dna_to_prot.py
```

1.2.5 Idea of solution

fill in

1.2.6 Analysation of results

fill in

You may notice that there are $4^3 = 64$ different codons, but only 20 amino acids. That is, some triplets encode the same amino acid.

1.3 Reverse translation

It has been observed that among the codons coding the same amino acid, some are more frequent than others. These frequencies can be converted to probabilities. E.g. consider codons `AUU`, `AUC`, and `AUA` that code for amino acid isoleucine. If they are observed, say, 36, 47, 17 times, respectively, to code isoleucine in a dataset, the probability that a random such event is `AUU` $\rightarrow$ isoleucine is 36/100.

This phenomenon is called *codon adaptation*, and for our purposes it works as a good introduction to generation of random sequences under constraints.

5. Consider the codon adaptation frequencies in http://www.kazusa.or.jp/codon/cgi-bin/showcodon.cgi?species=9606&aa=1&style=N and read them into a `dict()`, such that for each RNA sequence of length 3, say `AGU`, the hash table stores the probability of that codon among codons encoding the same amino acid. Put your solution in the `get_dict` function.

```
In [ ]: # exercise 5 DO NOT MODIFY THIS LINE
        %pycat part07-e05_codon_to_prob/src/codon_to_prob.py
```

1.3.1 Idea of solution

fill in

1.3.2 Analysation of results

fill in

Now you should have everything in place to easily solve the following.

6. Write a class `ProteinToMaxRNA` with a `convert` method which converts a protein sequence into the most likely RNA sequence to be the source of this protein. Run your program with `LTPIQNRA`.

```
In [ ]: # exercise 6 DO NOT MODIFY THIS LINE
        %pycat part07-e06_protein_to_max_rna/src/protein_to_max_rna.py
```

1.3.3 Idea of solution

fill in

1.3.4 Analysation of results

fill in

Now we are almost ready to produce random RNA sequences that code a given protein sequence. For this, we need a subroutine to *sample from a probability distribution*. Consider our earlier example of probabilities 36/100, 47/100, and 17/100 for `AUU`, `AUC`, and `AUA`, respectively. Let us assume we have a random number generator `random()` that returns a random number from interval $[0, 1)$. We may then partition the unit interval according to cumulative probabilities to [0,36/100), [36/100,83/100), [83/100,1), respectively. Depending which interval the number `random()` hits, we select the codon accordingly.

7. Write a function `random_event` that chooses a random event, given a probability distribution (set of events whose probabilities sum to 1). You can use function `random.uniform` to produce values uniformly at random from the range $[0, 1)$. The distribution should be given to your function as a dictionary from events to their probabilities.

```
In [ ]: # exercise 7 DO NOT MODIFY THIS LINE
        %pycat part07-e07_random_event/src/random_event.py
```

1.3.5 Idea of solution

fill in

1.3.6 Analysation of results

fill in

With this general routine, the following should be easy to solve.

8. Write a class `ProteinToRandomRNA` to produce a random RNA sequence encoding the input protein sequence according to the input codon adaptation probabilities. The actual conversion is done through the `convert` method. Run your program with `LTPIQNRA`.

```
In [ ]: # exercise 8 DO NOT MODIFY THIS LINE
        %pycat part07-e08_protein_to_random_rna/src/protein_to_random_rna.py
```

1.3.7 Idea of solution

fill in

1.3.8 Analysation of results

fill in

1.4 Generating DNA sequences with higher-order Markov chains

We will now reuse the machinery derived above in a related context. We go back to DNA sequences, and consider some easy statistics that can be used to characterize the sequences. First, just the frequencies of bases `A`, `C`, `G`, `T` may reveal the species from which the input DNA originates; each species has a different base composition that has been formed during evolution. More interestingly, the areas where DNA to RNA transcription takes place (coding region) have an excess

of `C` and `G` over `A` and `T`. To detect such areas a common routine is to just use a *sliding window* of fixed size, say k, and compute for each window position $T[i..i+k-1]$ the base frequencies, where $T[1..n]$ is the input DNA sequence. When sliding the window from $T[i..i+k-1]$ to $T[i+1..i+k]$ frequency $f(T[i])$ gets decreases by one and $f(T[i+k])$ gets increased by one.

9. Write a *generator* `sliding_window` to compute sliding window base frequencies so that each moving of the window takes constant time. We saw in the beginning of the course one way how to create generators using generator expression. Here we use a different way. For the function `sliding_window` to be a generator, it must have at least one `yield` expression, see https://docs.python.org/3/reference/expressions.html#yieldexpr. See also exercise 13 for an example of a generator `all_kmers` made with yield expressions. A yield expression can be used to return a value and *temporarily* return from the function.

```
In [ ]: # exercise 9 DO NOT MODIFY THIS LINE
        %pycat part07-e09_sliding_window/src/sliding_window.py
```

1.4.1 Idea of solution

fill in

1.4.2 Analysation of results

fill in

Our models so far have been so-called *zero-order* models, as each event has been independent of other events. With sequences, the dependencies of events are naturally encoded by their *contexts*. Considering that a sequence is produced from left-to-right, a *first-order* context for $T[i]$ is $T[i-1]$, that is, the immediately preceding symbol. *First-order Markov chain* is a sequence produced by generating $c = T[i]$ with the probability of event of seeing symbol c after previously generated symbol $a = T[i-1]$. The first symbol of the chain is sampled according to the zero-order model. The first-order model can naturally be extended to contexts of length k, with $T[i]$ depending on $T[i-k..i-1]$. Then the first k symbols of the chain are sampled according to the zero-order model. The following assignments develop the routines to work with the *higher-order Markov chains*. In what follows, a k-mer is a substring $T[i..i+k-1]$ of the sequence at an arbitrary position.

10. Write function `context_list` that given an input DNA sequence T associates to each k-mer W the concatenation of all symbols c that appear after context W in T, that is, $T[i..i+k] = Wc$. For example, `GA` is associated to `TCT` in $T =$ `ATGATATCATCGACGATGTAG`, when $k = 2$.

```
In [ ]: # exercise 10 DO NOT MODIFY THIS LINE
        %pycat part07-e10_context_list/src/context_list.py
```

1.4.3 Idea of solution

fill in

1.4.4 Analysation of results

fill in

11. With the above solution, write function `context_probabilities` to count the frequencies of symbols in each context and convert these frequencies into probabilities. Run your program with $T =$ `ATGATATCATCGACGATGTAG` and k values 0 and 2.

```
In [ ]: # exercise 11 DO NOT MODIFY THIS LINE
        %pycat part07-e11_context_probabilities/src/context_probabilities.py
```

1.4.5 Idea of solution

fill in

1.4.6 Analysation of results

fill in

12. With the above solution and the function `random_event` from the earlier exercise, write class `MarkovChain`. Its `generate` method should generate a random DNA sequence following the original k-th order Markov chain probabilities.

```
In [ ]: # exercise 12 DO NOT MODIFY THIS LINE
        %pycat part07-e12_generate_markov/src/generate_markov.py
```

1.4.7 Idea of solution

fill in

1.4.8 Analysation of results

fill in

If you have survived so far without problems, please run your program a few more times with different inputs. At some point you should get a lookup error in your hash-table! The reason for this is not your code, but the way we defined the model: Some k-mers may not be among the training data (input sequence T), but such can be generated as the first k-mer that is generated using the zero-order model.

A general approach to fixing such issues with incomplete training data is to use *pseudo counts*. That is, all imaginable events are initialized to frequency count 1.

13. Modify the previous solution 11 to use pseudo counts in order to obtain a k-th order Markov chain that can assign a probability for any DNA sequence. You may use the provided generator `all_kmers` to iterate over all k-mer of given length.

```
In [ ]: # exercise 13 DO NOT MODIFY THIS LINE
        %pycat part07-e13_pseudocounts/src/pseudocounts.py
```

1.4.9 Idea of solution

fill in

1.4.10 Analysation of results

fill in

14. Write class `MarkovChain` that given the k-th order Markov chain developed above to the constructor, its method `probability` computes the probability of a given input DNA sequence.

```
In [ ]: # exercise 14 DO NOT MODIFY THIS LINE
        %pycat part07-e14_markov_chain_probability/src/markov_chain_probability.py
```

1.4.11 Idea of solution

fill in

1.4.12 Analysation of results

fill in

With the last assignment you might end up in trouble with precision, as multiplying many small probabilities gives a really small number in the end. There is an easy fix by using so-called log-transform. Consider computation of $P = s_1 s_2 \cdots s_n$, where $0 \leq s_i \leq 1$ for each i. Taking logarithm in base 2 from both sides gives $\log_2 P = \log_2(s_1 s_2 \cdots s_n) = \log_2 s_1 + \log_2 s_2 + \cdots \log s_n = \sum_{i=1}^{n} \log s_i$, with repeated application of the property that the logarithm of a multiplication of two numbers is the sum of logarithms of the two numbers taken separately. The results is abbreviated as log-probability.

15. Write class `MarkovChain` that given the k-th order Markov chain developed above to the constructor, its method `log_probability` computes the log-probability of a given input DNA sequence. Run your program with $T =$ `ATGATATCATCGACGATGTAG` and $k = 2$.

```
In [ ]: # exercise 15 DO NOT MODIFY THIS LINE
        %pycat part07-e15_markov_chain_log_probability/src/markov_chain_log_probability.py
```

1.4.13 Idea of solution

fill in

1.4.14 Analysation of results

fill in

Finally, if you try to use the code so far for very large inputs, you might observe that the concatenation of symbols following a context occupy considerable amount of space. This is unnecessary, as we only need the frequencies.

16. Optimize the space requirement of your code from exercise 13 for the k-th order Markov chain by replacing the concatenations by direct computations of the frequencies. Implement this as the `context_probabilities` function.

```
In [ ]: # exercise 16 DO NOT MODIFY THIS LINE
        %pycat part07-e16_low_space_requirement/src/low_space_requirement.py
```

1.4.15 Idea of solution

fill in

1.4.16 Analysation of results

fill in

While the earlier approach of explicit concatenation of symbols following a context suffered from inefficient use of space, it does have a benefit of giving another much simpler strategy to sample from the distribution: observe that an element of the concatenation taken uniformly randomly is sampled exactly with the correct probability.

17. Revisit the solution 12 and modify it to directly sample from the concatenation of symbols following a context. You may use the function `random.choice`.

```
In [ ]: # exercise 17 DO NOT MODIFY THIS LINE
        %pycat part07-e17_sample_from_concatenation/src/sample_from_concatenation.py
```

1.4.17 Idea of solution

fill in

1.4.18 Analysation of results

fill in

1.5 k-mer index

Our k-th order Markov chain can now be modified to a handy index structure called k-mer index. This index structure associates to each k-mer its list of occurrence positions in DNA sequence T. Given a query k-mer W, one can thus easily list all positions i with $T[i..k-1] = W$.

18. Implement function `kmer_index` by modifying your earlier code for the k-th order Markov chain. Test your program with $T =$ `ATGATATCATCGACGATGTAG` and $k = 2$.

```
In [ ]: # exercise 18 DO NOT MODIFY THIS LINE
        %pycat part07-e18_kmer_index/src/kmer_index.py
```

1.5.1 Idea of solution

fill in

1.5.2 Analysation of results

fill in

1.6 Comparison of probability distributions

Now that we know how to learn probability distributions from data, we might want to compare two such distributions, for example, to test if our programs work as intended.

Let $P = \{p_1, p_2, \ldots, p_n\}$ and $Q = \{q_1, q_2, \ldots, q_n\}$ be two probability distributions for the same set of n events. This means $\sum_{i=1}^{n} p_i = \sum_{i=1}^{n} q_i = 1$, $0 \leq p_j \leq 1$, and $0 \leq q_j \leq 1$ for each event j.

Kullback-Leibler divergence is a measure $d()$ for the *relative entropy* of P with respect to Q defined as $d(P|Q) = \sum_{i=1}^{n} p_i \log \frac{p_i}{q_i}$.

This measure is always non-negative, and 0 only when $P = Q$. It can be interpreted as the gain of knowing Q to encode P. Note that this measure is not symmetric.

19. Write function `kullback_leibler` to compute $d(P||Q)$. Test your solution by generating a random RNA sequence encoding the input protein sequence according to the input codon adaptation probabilities. Then you should learn the codon adaptation probabilities from the RNA sequence you generated. Then try the same with uniformly random RNA sequences (which don't have to encode any specific protein sequence). Compute the relative entropies between the three distribution (original, predicted, uniform) and you should observe a clear difference. Because $d(P||Q)$ is not symmetric, you can either print both $d(P||Q)$ and $d(Q||P)$, or their average.

```
In [ ]: # exercise 19 DO NOT MODIFY THIS LINE
        %pycat part07-e19_kullback_leibler/src/kullback_leibler.py
```

1.6.1 Idea of solution

fill in

1.6.2 Analysation of results

fill in

1.7 Stationary and equilibrium distributions

Let us consider a Markov chain of order one on the set of nucleotides. Its transition probabilities can be expressed as a 4×4 matrix $P = (p_{ij})$, where the element p_{ij} gives the probability of the jth nucleotide on the condition the previous nucleotide was the ith. An example of a transition matrix is

	A	C	G	T
A	0.30	0.0	0.70	0.0
C	0.00	0.4	0.00	0.6
G	0.35	0.0	0.65	0.0
T	0.00	0.2	0.00	0.8

.

A distribution $\pi = (\pi_1, \pi_2, \pi_3, \pi_4)$ is called *stationary*, if $\pi = \pi P$ (the product here is matrix product).

20. Write function `get_stationary_distributions` that gets a transition matrix as parameter, and returns the list of stationary distributions. You can do this with NumPy by first taking transposition of both sides of the above equation to get equation $\pi^T = P^T \pi^T$. Using

numpy.linalg.eig take all eigenvectors related to eigenvalue 1.0. By normalizing these vectors to sum up to one get the stationary distributions of the original transition matrix. In the `main` function print the stationary distributions of the above transition matrix.

```
In [ ]: # exercise 20 DO NOT MODIFY THIS LINE
        %pycat part07-e20_stationary_distribution/src/stationary_distribution.py
```

1.7.1 Idea of solution

fill in

1.7.2 Analysation of results

fill in

21. Implement the following in the `main` function. Use again the above transition matrix. Choose one stationary distribution as the initial distribution of the Markov chain. Using your modified Markov chain generator generate a nucleotide sequence s of length 10~000. Choose prefixes of s of lengths 1, 10, 100, 1000, and 10 000. For each of these prefixes find out their nucleotide distribution (of order 0) using your earlier tool. Use 1 as the pseudo count. Then, for each prefix, compute the KL divergence between the initial distribution and the normalized nucleotide distribution.

```
In [ ]: # exercise 21 DO NOT MODIFY THIS LINE
        %pycat part07-e21_stationary_initial_distribution/src/stationary_initial_distribution.p
```

1.7.3 Idea of solution

fill in

1.7.4 Analysation of results

fill in

22. Implement the following in the `main` function. Find the stationary distribution for the following transition matrix:

	A	C	G	T
A	0.30	0.10	0.50	0.10
C	0.20	0.30	0.15	0.35
G	0.25	0.15	0.20	0.40
T	0.35	0.20	0.40	0.05

Since there is only one stationary distribution, it is called the *equilibrium distribution*. Choose randomly two nucleotide distributions. You can take these from your sleeve or sample them from the Dirichlet distribution. Then for each of these distributions as the initial distribution of the Markov chain, repeat the above experiment. The state distribution should converge to the equilibrium distribution no matter how we start the Markov chain!

```
In [ ]: # exercise 22 DO NOT MODIFY THIS LINE
        %pycat part07-e22_equilibrium_distribution/src/equilibrium_distribution.py
```

1.7.5 Idea of solution

fill in

1.7.6 Analysation of results

fill in